OPERATING SYSTEM

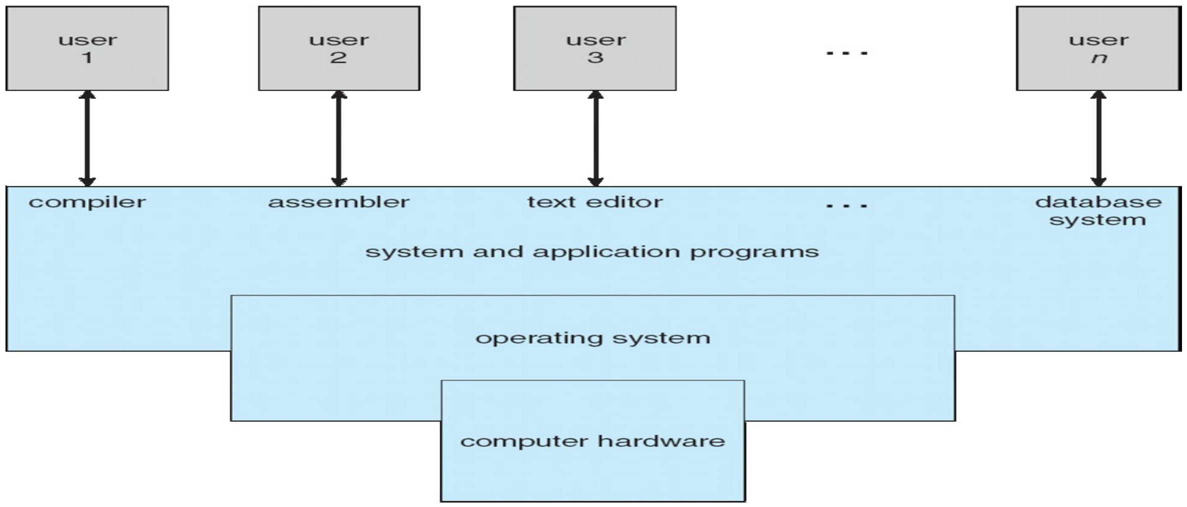
What is an Operating System?
A program that acts as an intermediary between a user of a computer and the computer hardware
Operating system goals:

- Execute user programs and make solving user problems easier
- Make the computer system convenient to use
- Use the computer hardware in an efficient manner

Computer System Structure
Computer system can be divided into four components

- Hardware – provides basic computing resources
  4CPU, memory, I/O devices
- Operating system
  4Controls and coordinates use of hardware among various applications and users
- Application programs – define the ways in which the system resources are used to solve the computing problems of the users
  4Word processors, compilers, web browsers, database systems, video games
- Users
  4People, machines, other computers

Four Components of a Computer System



User View of OS

CASE1

OS is designed for ease of use

- Performance
  - Important to the user
- Resource utilization

CASE2

User accessing mainframe or minicomputer

- Other users are accessing the same computer
- Users share resources and may exchange information
- Resource utilization

CASE3

Users using workstations

- Dedicated resource
- OS is designed to compromise between individual usability and resource utilization

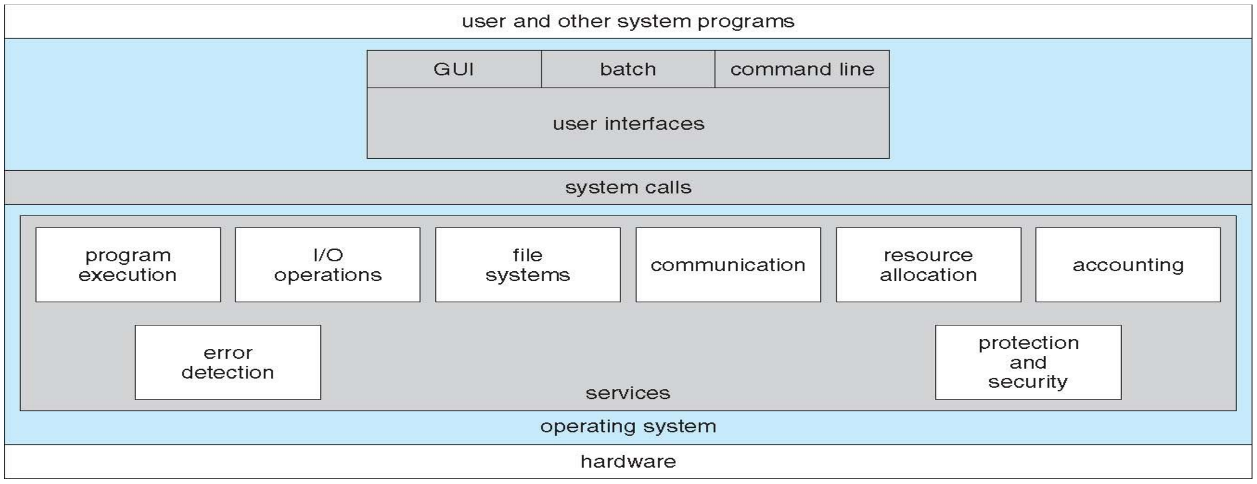
CASE4 (Recent)

OS is designed for individual usability, but performance per amount of battery life is important as well.

System view of OS

- OS is a **resource allocator**
  Required in many users using sharing Mainframe, Manages all resources, Decides between conflicting requests for efficient and fair resource use
- OS is a **control program**
  Controls execution of programs to prevent errors and improper use of the computer, Concerned with the operation and control of I/O devices

A View of Operating System Services



Operating System Services

One set of operating-system services provides functions that are helpful to the user:

1. **User interface** - Almost all operating systems have a user interface (UI)
   a. 4Varies between Command-Line (CLI), Graphics User Interface (GUI), Batch
2. **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
3. **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
4. **File-system manipulation** - The file system is of particular interest. Obviously, programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.

5. **Communications** – Processes may exchange information, on the same computer or between computers over a network
   a. 4Communications may be via shared memory or through message passing (packets moved by the OS)
6. **Error detection** – OS needs to be constantly aware of possible errors
   a. 4May occur in the CPU and memory hardware, in I/O devices, in user program
   b. 4For each type of error, OS should take the appropriate action to ensure correct and consistent computing
   c. 4Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

   d. Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
7. **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
   a. 4Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
8. **Accounting -** To keep track of which users use how much and what kinds of computer resources
9. **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other
   a. **Protection** involves ensuring that all access to system resources is controlled
   b. **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts
   c. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

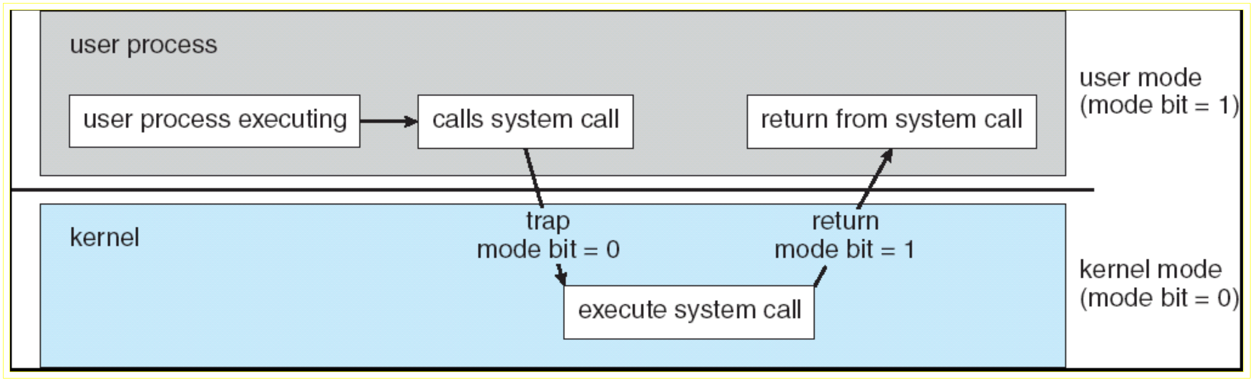
Operating System Functions

Dual-mode operation allows OS to protect itself and other system components

User mode and kernel mode

Mode bit provided by hardware

- Provides ability to distinguish when system is running user code or kernel code
- Some instructions designated as privileged, only executable in kernel mode
- System call changes mode to kernel, return from call resets it to user



System call

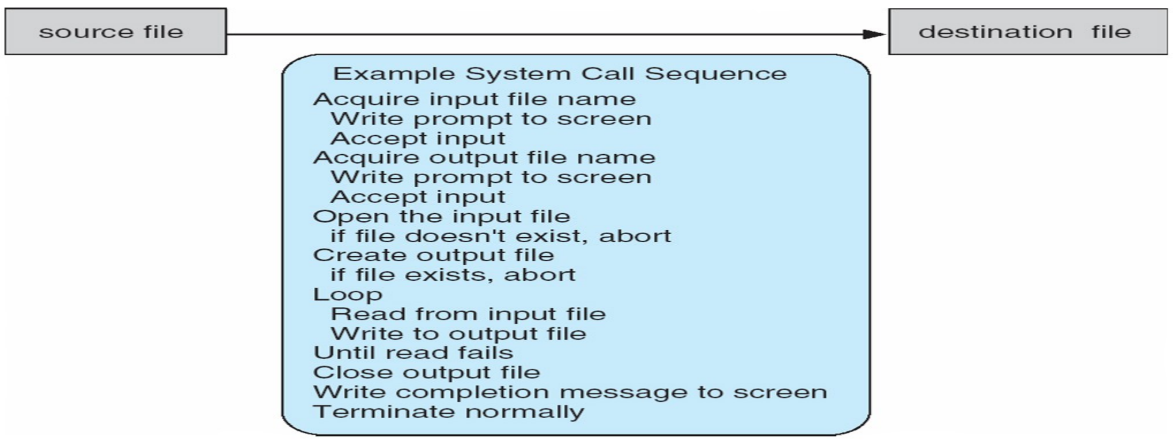
System Call is a programming interface to the services provided by the OS. Typically written in a high-level language (C or C++). Mostly accessed by programs via a high-level Application Program Interface (API) rather than direct system call use

Three most common APIs are

i.Win32 API for Windows,

ii. POSIX API for POSIX-based systems (including UNIX, Linux, and Mac OS X)

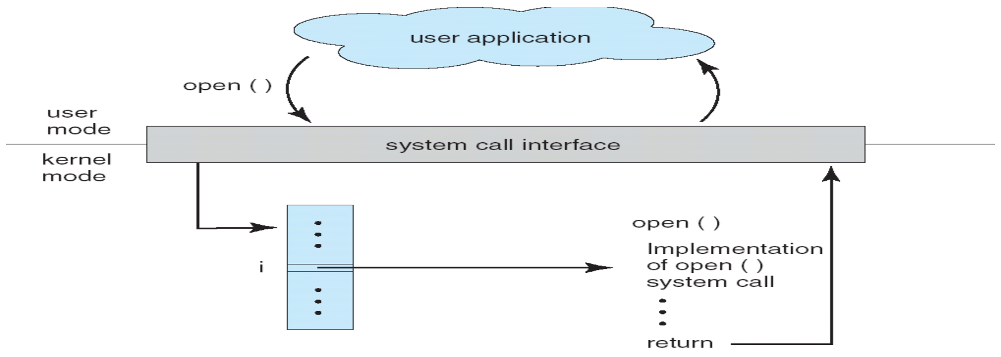
iii. Java API for the Java Virtual Machine (JVM)



```
Acquire input file name
  Write prompt to screen
  Accept input
Acquire output file name
  Write prompt to screen
  Accept input
Open the input file
  if file doesn't exist, abort
Create output file
  if file exists, abort
Loop
  Read from input file
  Write to output file
Until read fails
Close output file
Write completion message to screen
Terminate normally
```

System Call Implementation

- Typically, a number associated with each system call. System-call interface maintains a table indexed according to these Numbers, The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
- Just needs to obey API and understand what OS will do as a result call
- Most details of OS interface hidden from programmer by API
  4Managed by run-time support library (set of functions built into libraries included with compiler



System calls can be grouped roughly into five major categories:

1) Process control
2) File management
3) Device management
4) Information maintenance
5) Communications

Process Control
A running program needs to be able to halt its execution either normally (end) or abnormally (abort). Under either normal or abnormal circumstances, the operating system must transfer control to the invoking **command interpreter**.
The command interpreter then reads the next command.
In an interactive system, the command interpreter simply continues with the next command.
In a GUI system, a pop-up window might alert the user to the error and ask for guidance.
In a batch system, the command interpreter usually terminates the entire job and continues with the next job.
If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process job's priority, its maximum allowable execution time, and so on (get process attributes and set process attributes). After creating new jobs or processes, we may need to wait for them to finish their execution.

- wait for a certain amount of time to pass (wait time);
- wait for a specific event to occur (wait event).

The jobs or processes should then signal when that event has occurred (signal event).

File management
Information that need to be saved "permanently" must be stored in a secondary storage device e.g. a disk, tape, etc. Files are an abstraction of secondary storage devices
File manager is responsible for **create file, delete file open, close ,read, write, reposition (rewinding or skipping to the end of the file) get file attributes, set file attributes Filename, file type, protection codes, accounting information**

Device management
A process may need several resources to execute main memory, disk drives, access to files. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.
Functions

request device, release device
read, write, reposition
get device attributes, set device attributes
logically attach or detach devices

Information Maintenance
Many system calls exist simply for the purpose of transferring information between the User program and Operating system. For example, most systems have a system call to return the current time and date. Other system calls may return information about the system, such as

the number of current users,
the version number of the operating system,
the amount of free memory or disk space, and so on.

The operating system keeps information about all its processes, and system calls are used to access this information. Generally, calls are also used to reset the process information (get process attributes and set process attributes).

Communication

There are two common models of interprocess communication:

a. Message- passing model and
b. Shared-memory model.

Message-passing model

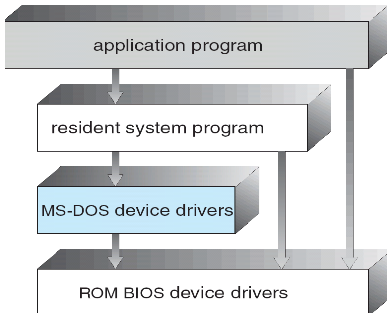
c. The communicating processes exchange messages with one another to transfer information.
d. Messages can be exchanged between the processes either directly or indirectly through a common mailbox.
e. Before communication can take place, a connection must be opened.
f. The name of the other communicator must be known, be it another
    - process on the same system or a
    - process on another computer connected by a communications network

Operating System Structure

Simple Structure

MS-DOS – written to provide the most functionality in the least space, Not divided into modules Its interfaces and levels of functionality are not well separated.For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives.
Example :

MS-DOS Layer Structure



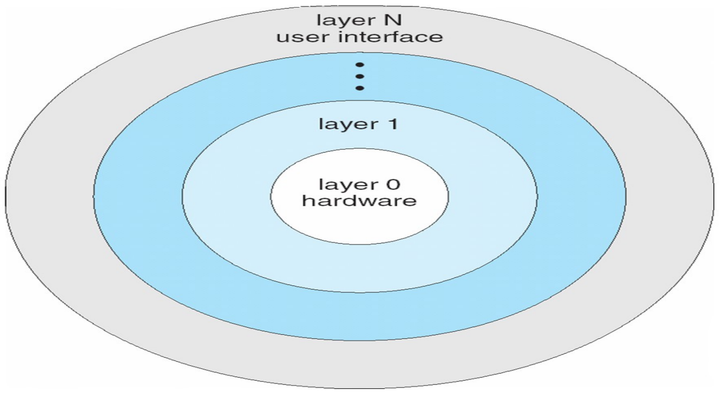
Example:

UNIX

- UNIX – limited by hardware functionality, the original UNIX operating system had limited structuring. The UNIX OS consists of two separable parts
- Systems programs
- The kernel
  4Consists of everything below the system-call interface and above the physical hardware
  4Provides the file system, CPU scheduling, memory management, and other operating-system functions; a large number of functions for one level

Layered Approach

- The operating system is divided into a number of layers (levels), each built on top of lower layers. The bottom layer (layer 0), is the hardware; the highest (layer N) is the user interface.
- With modularity, layers are selected such that each uses functions (operations) and services of only lower-level layers



Micro kernel System Structure

Microkernel method structures the operating system by removing

- all nonessential components from the kernel and
- implementing them as system and user-level programs.

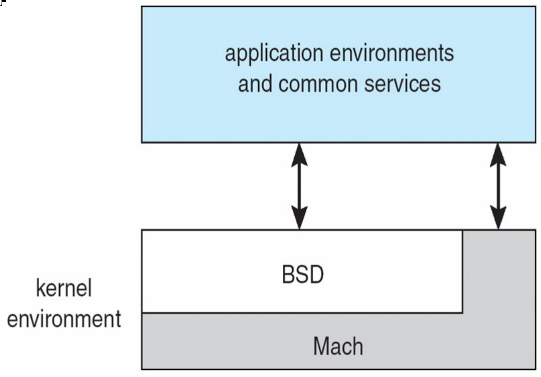
The result is a smaller kernel. Microkernel provide minimal process and memory management
The main function of the microkernel is to provide a communication facility between the client program and the various services that are also running in user space.
Communication takes place between user modules using message passing
Benefits:

- Easier to extend a microkernel
- Easier to port the operating system to new architectures
- More reliable (less code is running in kernel mode)
- More secure

Detriments:

- Performance overhead of user space to kernel space communication.

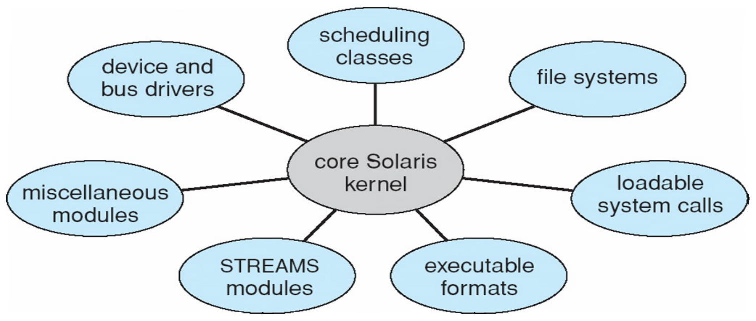
Example : Mac OS X Structure



Modules

- Most modern operating systems implement kernel modules
- Uses object-oriented approach
- Each core component is separate
- Each talks to the others over known interfaces
- Each is loadable as needed within the kernel
- Overall, similar to layers but with more flexible

Example: Solaris Module



This design allows the kernel to provide core services yet also allows certain features to be implemented dynamically. For example, device and bus drivers for specific hardware can be added to the kernel, and support for different file systems can be added as loadable modules. The overall result resembles a layered system in that each kernel section has defined, protected interfaces; but it is more flexible than a layered system in that any module can call any other module.Primary module has only core functions and knowledge of how to load and communicate with other modules; but it is more efficient, because modules do not need to invoke message passing in order to communicate.

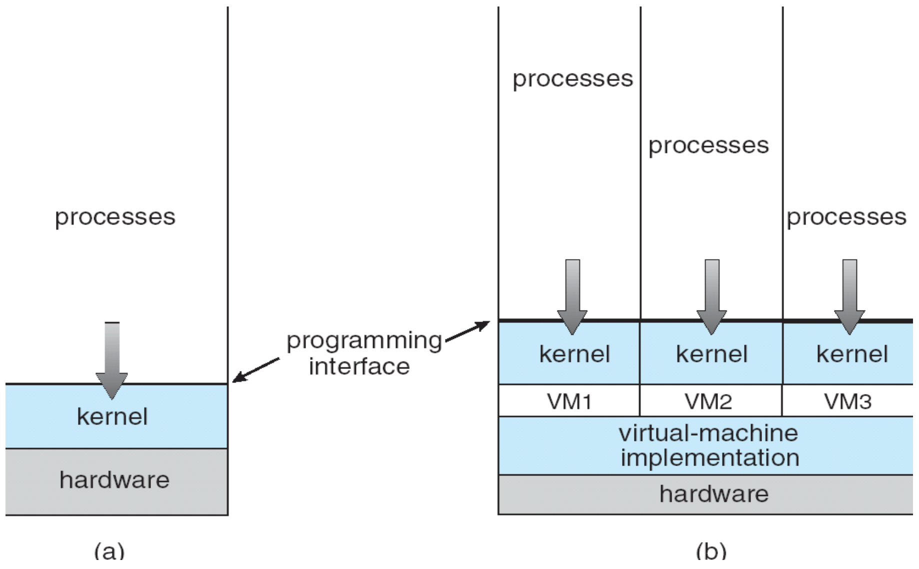
Virtual Machines

- A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware
- A virtual machine provides an interface *identical* to the underlying bare hardware

- The operating system host creates the illusion that a process has its own processor and (virtual memory)
- Each guest provided with a (virtual) copy of underlying computer

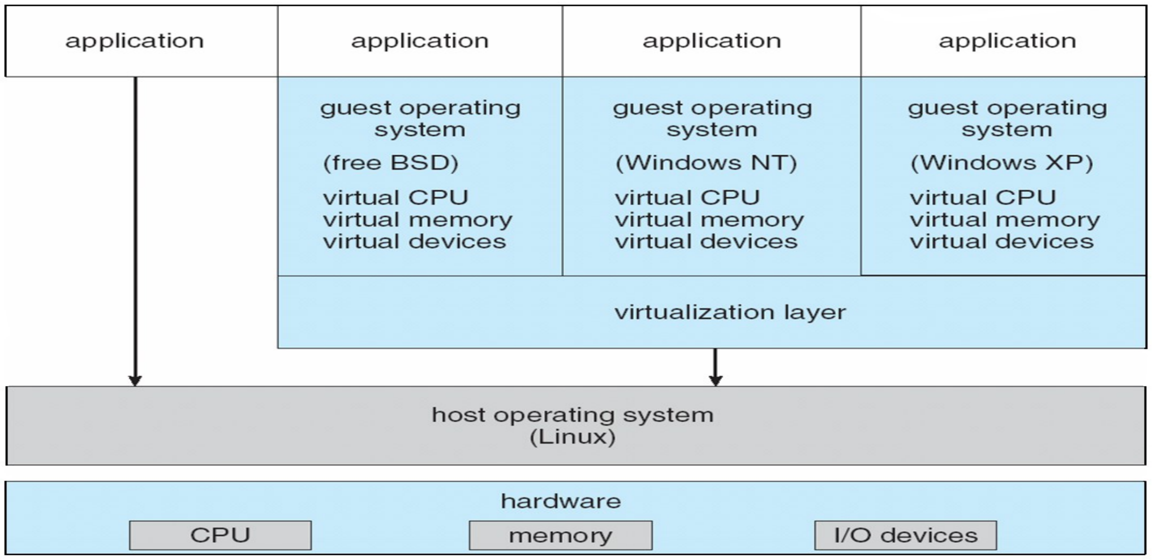
Virtual Machines History and Benefits

- First appeared commercially in IBM mainframes in 1972
- Fundamentally, multiple execution environments (different operating systems) can share the same hardware
- Protect from each other
- Some sharing of file can be permitted, controlled
- Commutate with each other, other physical systems via networking
- Useful for development, testing
- Consolidation of many low-resource use systems onto fewer busier systems
- "Open Virtual Machine Format", standard format of virtual machines, allows a VM to run within many different virtual machine (host) platforms

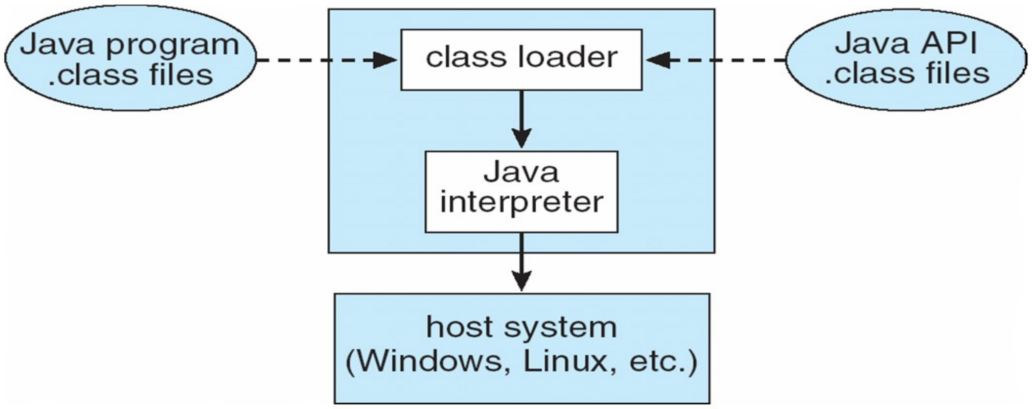


Example 1:
VMware Architecture
VMware is a popular commercial application that abstracts Intel 80x86 hardware into isolated virtual machines. VMware runs as an application on a host OS such as Windows or Linux and

allows this host system to concurrently run several different guest OS as independent virtual machine. Consider the following scenario: A developer has designed an application and would like to test it on Linux, FreeBSD, Windows NT, and Windows XP.One option is for her to obtain four different computers, each running a copy of one of these operating systems. Another alternative is for her first to install Linux on a computer system and test the application, then to install FreeBSD and test the application, and so forth.In this case, the programmer could test the application on a host operating system and on three guest operating systems with each system running as a separate virtual machine.



Example 2
The Java Virtual Machine
Java is a popular object-oriented programming language introduced by Sun Microsystems in 1995. In addition to a language specification and a large API library Java also provides a specification for a Java virtual machine—or JVM. The JVM is a specification for an abstract computer. It consists of a class loader and a Java interpreter that executes the architecture neutral bytecodes. The class loader loads the compiled .class files from both the Java program and the Java API for execution by the Java interpreter. After a class is loaded, the verifier checks that the .class file is valid Java bytecode and does not overflow or underflow the stack. It also ensures that the bytecode does not perform pointer arithmetic, which could provide illegal memory access. If the class passes verification, it is run by the Java interpreter. The JVM also automatically manages memory by performing garbage collection—the practice of reclaiming



Operating-System Debugging

Debugging is finding and fixing errors, or bugs.OS generate log files containing error information

Operating System Generation

- Operating systems are designed to run on any of a class of machines; the system must be configured for each specific computer site
- SYSGEN program obtains information concerning the specific configuration of the hardware system
- *Booting* – starting a computer by loading the kernel
- *Bootstrap program* – code stored in ROM that is able to locate the kernel, load it into memory, and start its execution

System Boot

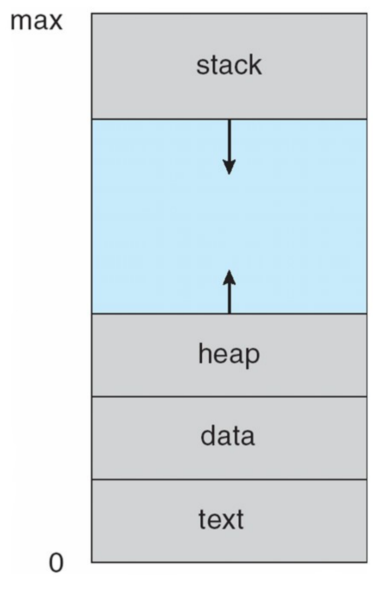
- Operating system must be made available to hardware so hardware can start it
- Small piece of code – **bootstrap loader**, locates the kernel, loads it into memory, and starts it
- Sometimes two-step process where **boot block** at fixed location loads bootstrap loader
- When power initialized on system, execution starts at a fixed memory location
  4Firmware used to hold initial boot code

Process Concept

Process – a program in execution; process execution must progress in sequential fashion
A process includes:

- **program counter**
- **stack**
- **data section**

Process in Memory

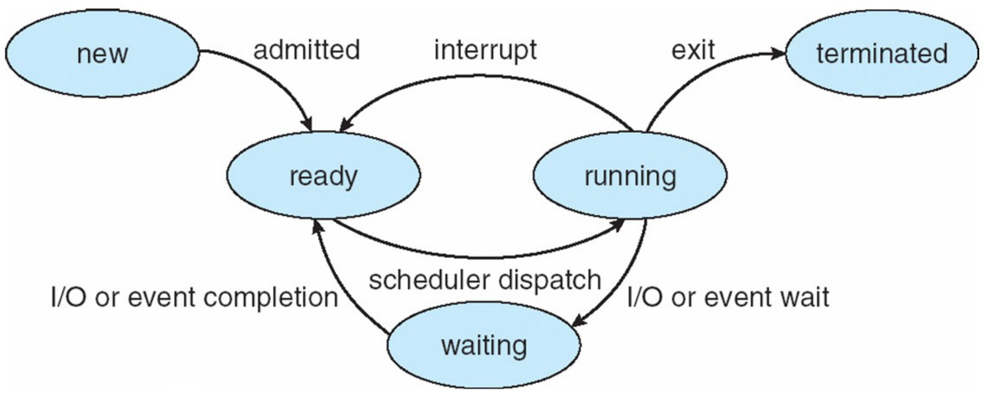


Process State

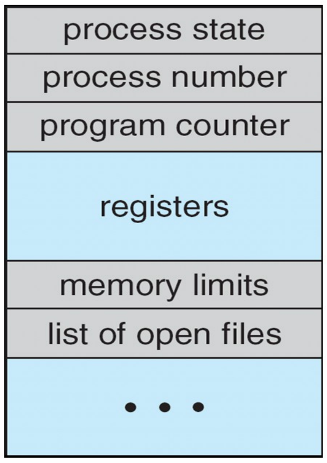
As a process executes, it changes *stat*
e

1) **new**: The process is being created
2) **running**: Instructions are being executed
3) **waiting**: The process is waiting for some event to occur
4) **ready**: The process is waiting to be assigned to a processor
5) **terminated**: The process has finished execution

Diagram of Process State



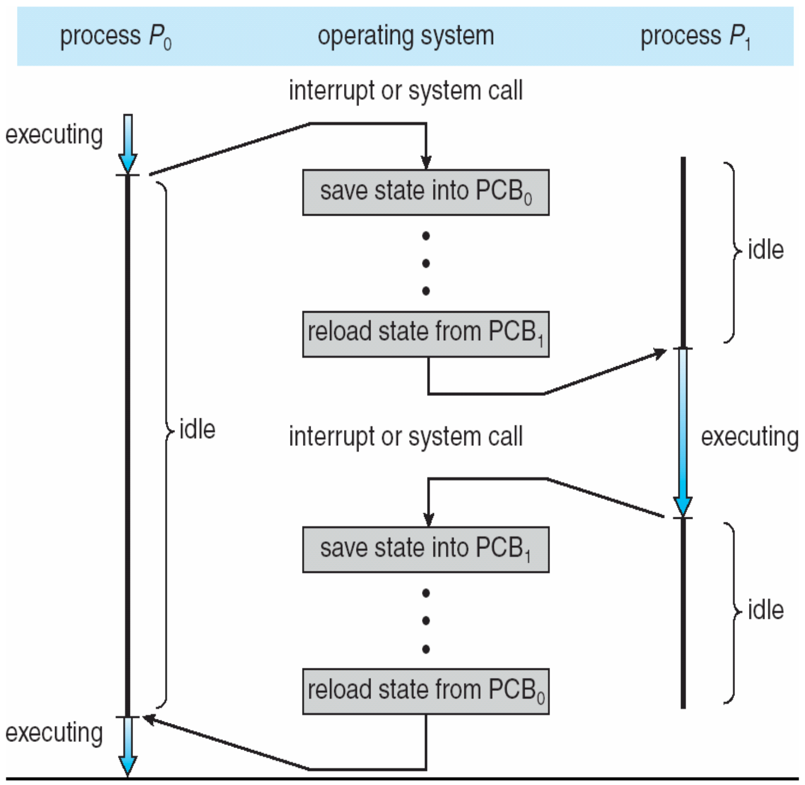
Process Control Block (PCB)



Information associated with each process

- Process state
- Program counter
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status information

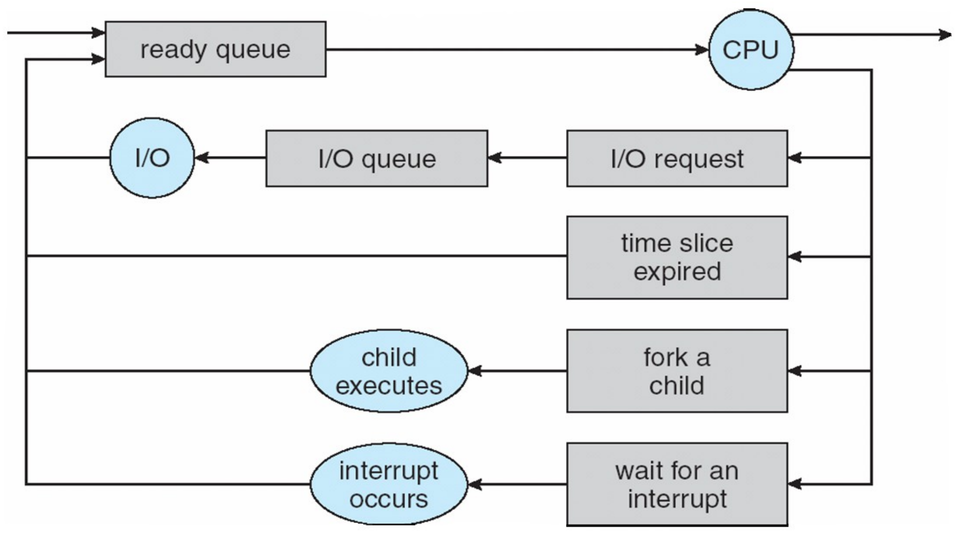
CPU Sv



Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues.

Representation of Process Scheduling



Schedulers

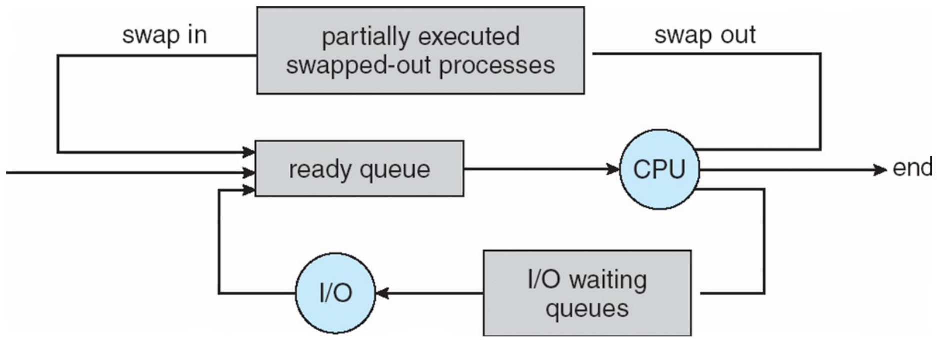
- **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
- **Short-term scheduler** (or CPU scheduler) – selects which process should be executed next and allocates CPU

Short-term scheduler is invoked very frequently (milliseconds) Þ (must be fast)
Long-term scheduler is invoked very infrequently (seconds, minutes) Þ (may be slow).The long-term scheduler controls the *degree of multiprogramming*

Processes can be described as either:

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts
- **CPU-bound process** – spends more time doing computations; few very long CPU bursts

Addition of Medium Term Scheduling



It removes the processes from memory (and from active contention for the CPU) and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix.

Context Switch

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch. Context of a process represented in the PCB
- Context-switch time is overhead; the system does no useful work while switching
- Time dependent on hardware support

Interprocess Communication (IPC)

Processes executing concurrently in the operating system may be either independent processes or cooperating processes.

Independent process cannot affect or be affected by the execution of another process
It does not share data with any other process
Cooperating process can affect or be affected by the execution of another process.It shares data with other processes

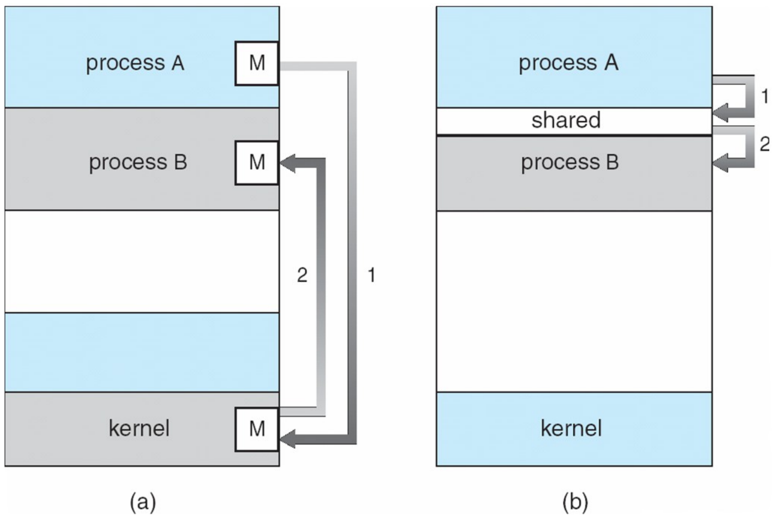
Advantages of process cooperation
Information sharing.
Several users may be interested in the same piece of information . Provide an environment to allow concurrent access to such information.

Computation speedup.
If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements

Modularity.
We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads
Convenience.
Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, and compiling in parallel.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication:
Shared Memory and Message Passing.
In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
In the message- passing model, communication takes place by means of messages exchanged between the cooperating processes.

Communications Models



Shared memory systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, a shared-memory region resides in the address space of the process creating the shared-memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously. So Here cooperation is required.

Examples: **Producer –Consumer Problem**
compiler-assembler-loader.
client—server paradigm
web server-web browser

One solution to the producer—consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a **buffer** of items that can be filled by the producer and emptied by the consumer.

Interprocess Communication – Message Passing

Message system – processes communicate with each other without resorting to shared variables
IPC facility provides two operations:

send(*message*) – message size fixed or variable
receive(*message*)

If *P* and *Q* wish to communicate, they need to: establish a *communication link* between them, exchange messages via send/receive
Implementation of communication link

physical (e.g., shared memory, hardware bus)
logical (e.g., logical properties)

Several methods for logically implementing a link and the send/receive

Direct or indirect communication
Synchronous or asynchronous communication
Automatic or explicit buffering

Direct Communication

Processes must name each other explicitly:

send (*P, message*) – send a message to process P
receive(*Q, message*) – receive a message from process Q

Properties of communication link are

Links are established automatically
A link is associated with exactly one pair of communicating processes
Between each pair there exists exactly one link
The link may be unidirectional, but is usually bi-directional

Indirect communication

Messages are directed and received from mailboxes (also referred to as ports)
Each mailbox has a unique id
Processes can communicate only if they share a mailbox

Properties of communication links are
Link established only if processes share a common mailbox
A link may be associated with many processes
Each pair of processes may share several communication links
Link may be unidirectional or bi-directional

Operations
create a new mailbox
send and receive messages through mailbox
destroy a mailbox

Primitives are defined as:
send(*A, message*) – send a message to mailbox A
receive(*A, message*) – receive a message from mailbox A

Mailbox sharing
P_1, P_2, and P_3 share mailbox A
P_1, sends; P_2 and P_3 receive
Who gets the message?

Solutions
Allow a link to be associated with at most two processes
Allow only one process at a time to execute a receive operation
Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking. **Blocking** is considered **synchronous**
- **Blocking send** has the sender block until the message is received
- **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous .Non-blocking** send has the sender send the message and continue
- **Non-blocking** receive has the receiver receive a valid message or null

Buffering

Queue of messages attached to the link; implemented in one of three ways
1. Zero capacity – 0 messages
Sender must wait for receiver (rendezvous)

2. Bounded capacity – finite length of *n* messages
Sender must wait if link full
3. Unbounded capacity – infinite length
Sender never waits

Multithreaded programming

Threads

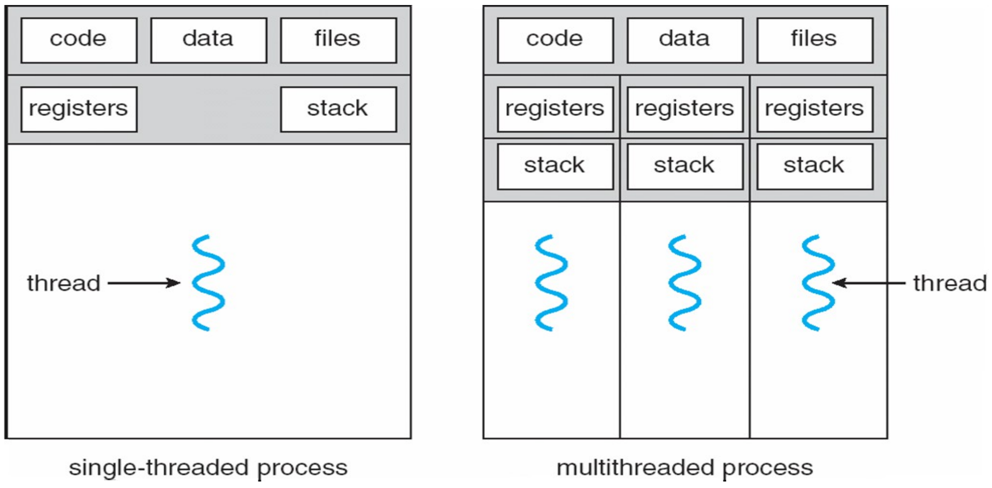
A thread is a basic unit of CPU utilization; it comprises a

thread ID,
a program counter,
a register set, and
a stack.

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources. A traditional (or heavyweight) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

Many software packages that run on modern desktop PCs are multithreaded.
An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network. A word processor may have a one thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background**.**

Single and Multithreaded Processes



Benefits

- **Responsiveness**
    - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
    - For instance, a multithreaded web browser.
- **Resource sharing**.
    - By default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows

an application to have several different threads of activity within the same address space

- **Economy.**
    - Allocating memory and resources for process creation is costly.
    - Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads.
    - It is much more time consuming to create and manage processes than threads.

- **Utilization of multiprocessor architectures. (Scalability)**

  Here threads may be running in parallel on different processors. A single-threaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency

User Threads

Thread management done by user-level threads library

- . Three primary thread libraries:

  i) POSIX Pthreads
- ii) Win32 threads iii) Java threads

Kernel Threads

Supported by the Kernel

Examples

Windows XP/2000, Solaris, Linux, Mac OS X

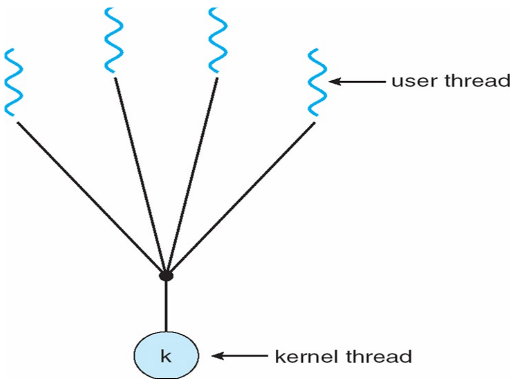
Multithreading Models

a) Many-to-One
b) One-to-One
c) Many-to-Many

Many-to-One

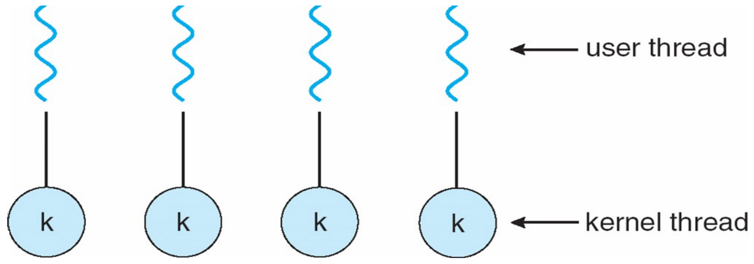
Many user-level threads mapped to single kernel thread .Thread management is done by the thread library in user space, so it is **efficient; but the entire process will block if a thread makes a blocking system call.** Also, because only one thread can access the kernel at a time, **multiple threads are unable** to run in parallel on multiprocessors.

Examples:
Solaris Green Threads
GNU Portable Threads



One-to-One

It provides **more concurrency** than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows **multiple threads to run in parallel** on multiprocessors. The only **drawback to this model is that creating a user thread requires creating the corresponding kernel thread.** Because the overhead of creating kernel threads can **burden the performance of an application**, most implementations of this model restrict the number of threads supported by the system.



Examples

Windows NT/XP/2000
Linux
Solaris 9 and later

Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine .Allows the operating system to create a sufficient number of kernel threads. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.

Examples : Windows NT/2000 with the *ThreadFiber* package

Thread Libraries

Thread library provides programmer with API for creating and managing threads. Two primary ways of implementing

First approach

- It provides a library entirely in user space with **no kernel support**.
- All code and data structures for the library exist in **user space**.
- This means that invoking a function in the library results in a **local function call** in user space and **not a system call**.

Second approach

- Implement a kernel-level library supported directly by the **operating system**.
- In this case, code and data structures for the library exist in **kernel space**.
- Invoking a function in the API for the library typically results in a **system call** to the kernel.

Pthreads

- May be provided either as user-level or kernel-level .A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.API specifies behavior of the thread library, implementation is up to development of the library Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Java Threads

- Java threads are managed by the JVM.Typically implemented using the threads model provided by underlying OS.Java threads may be created by Extending Thread class and second approach is by implementing the Runnable interface

Threading Issues

1. Thread cancellation of target thread
2. Signal handling
3. Thread pools
4. Thread-specific data
5. Scheduler activations
6. Semantics of **fork()** and **exec()** system calls

Thread Cancellation

Terminating a thread before it has finished. Two general approaches:
Asynchronous cancellation terminates the target thread immediately
Deferred cancellation allows the target thread to periodically check if it should be cancelled

Signal Handling

Signals are used in UNIX systems to notify a process that a particular event has occurred
A signal handler is used to process signals

1.Signal is generated by particular event
2.Signal is delivered to a process
3.Signal is handled

Options:

Deliver the signal to the thread to which the signal applies
Deliver the signal to every thread in the process
Deliver the signal to certain threads in the process
Assign a specific thread to receive all signals for the process

Thread Pools

Create a number of threads in a pool where they await work
Advantages: Usually slightly faster to service a request with an existing thread than create a new thread
Allows the number of threads in the application(s) to be bound to the size of the pool

Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application. Scheduler activations provide up calls - a communication mechanism from the kernel to the thread library
  This communication allows an application to maintain the correct number kernel threads

Semantics of fork() and exec() system calls

The semantics of the **fork()** and **exec()** system calls change in a multithreaded program.Does **fork()** duplicate only the calling thread or all threads? Some UNIX systems have chosen to have two versions of **fork()**, one that duplicates all threads and another that duplicates only the thread that invoked the fork () system call.